3.4 Solution:
There are many ways to make this notion precise. Here is one way. An enumerator
is a 5-tuple \((Q, \Gamma, \Sigma, \delta, q_{\text{initial}})\), where \(Q\), \(\Sigma\), \(\Gamma\) are all finite sets and

1. \(Q\) is the set of states
2. \(\Gamma\) is the first tape alphabet; it contains blank but not #
3. \(\Sigma\) is the alphabet, containing neither # nor blank, over which the language to be
generated is defined, and the second tape alphabet is \(\Sigma'\), which is \(\Sigma\) together with
\# and blank
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\} \times ((\Sigma' \times \{R\}) \cup (\{\varepsilon\} \times \{S\}))\) is the transition
function
5. \(q_{\text{initial}} \in Q\) is the start state

Initially both tapes are blank and the machine is in state \(q_{\text{initial}}\). When \(\delta(q,x) =
(q',y,D,z,M)\), if the machine is in state \(q\) and the first tape head is reading symbol \(x\),
then the machine enters state \(q'\); writes a \(y\) under the first tape head; moves the first
tape head left or right according to direction \(D\); and if \(M=R\) it writes a \(z\) under the
second tape head and moves the second tape head right. (When \(M=R\), no write is
performed on the second tape and the tape head does not move.)

A string \(w\) over the alphabet \(\Sigma\) is said to be generated by the machine if after a finite
(but arbitrarily large) number of steps the string \(#w#\) appears on the second tape. The
language generated by the machine is the set of all strings generated by the machine.

3.7 Solution:
The description is not legitimate because step 1 must enumerate all possible settings
of \(x_1, \ldots, x_k\), but there is an infinite number of settings. So \(M_{\text{had}}\) will never complete
this step and not be able to go on running steps 2 and 3. (Every step has to involve
operations that each take a finite amount of work.)

#3 Solution:
Consider the language \(L = \{w \mid w \text{ contains exactly twice as many } 0\text{'s as } 1\text{'s}\}\) over the
alphabet \(\{0,1\}\). Give an implementation-level description and a low-level
description of a Turing machine that decides \(L\).

Solution:
(1) Implementation-level description:
\(M = \text{“On input } w:\)\
   I. Scan \(w\), if a ‘1’ is found, cross it out, and move the head of the tape back to the
left-hand end, and then go to II. If there’s no ‘1’ left and no ‘0’ left, go to IV; else
   go to V.
   II. Scan \(w\), if a ‘0’ is found, cross it out, and go to stage III. If there’s no ‘0’ left,
go to stage V.
III. Continue to scan w, if another ‘0’ is found, cross it out and move the head of the tape to the left-hand end, and then go to stage I. If there’s no ‘0’ left, go to stage V.
IV. Accept.
V. Reject.”

(2) Low-level description:
We construct a Turing machine with 11 states \{1,2,3,4,5,6,7,8,9,Accept,Reject\} having initial state 1, with \(\Sigma = \{0,1\}\), and \(\Gamma = \{0,1,\#,X,e\}\). The transition function is given in the following table, where columns specify the tape symbol being read and rows indicate the current state. Each entry gives the new state, the tape symbol to write, and the direction to move the tape head. States 1,2,3 are used to write a # at the beginning of the input (unless it is empty, in which case we accept straight off). State 4 scans back to the beginning of the tape. Then states 5 and 6 scan for a 1 and change it to an X; if no 1 is found but we saw a 0, we hit the blank in state 6, but if no 0 was seen we hit it in state 5. State 7 scans back to the beginning of the tape. State 8 finds the first 0 and crosses it off. State 9 finds the second and crosses it off, going back to state 4 to “rewind” the tape.
A state diagram could be drawn instead.

![Tape Symbol Transition Table](image)

3.13 Solution:
We show that the modified Turing machine is no more powerful than a DFA. The transition function of the modified Turing machine \(M\) is defined as \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R,S\}\). First we show that we can eliminate transitions that do not move the tape head. To do this, whenever \(\delta(q,x) = (q’,y,S)\) and \(\delta(q’,y) = (q’’,z,S)\) we note that whenever the first transition is used it is forced to be followed by the second. So change the transition \(\delta(q,x) = (q’,y,S)\) to \(\delta(q,x) = (q’’,z,S)\). If in the process we ever form a transition \(\delta(q,x) = (q,x,S)\), this will cause an infinite loop if it is ever executed, so set \(\delta(q,x) = \text{reject}\) instead (the TM cannot accept if it ever does
this anyway). Now there cannot be two stay put transitions in a row. Next whenever \( \delta(q,x) = (q',y,S) \) and \( \delta(q',y) = (q'',z,R) \), change the transition \( \delta(q,x) = (q',y,S) \) to \( \delta(q,x) = (q'',z,R) \). Note that now ALL transitions move right – none stay put.

Because every transition moves right after writing, nothing written on the tape is ever read again, and hence tape can be made read-only. For each transition \( \delta(q,x) = (q'',z,R) \), we take \( \delta'(q,x) = q'' \) to (nearly) form the transition function of a DFA. We must be careful about a few other things. In the accept state of the TM (which is also the accept state of the DFA), we add transitions in the FA so that any non-blank symbol can be read while staying in the accept state and moving right. We do the same for the reject state. There is one more issue, that the TM may move right forever. To avoid this, we change the description of the TM so that \( \delta(q,\text{blank}) = (\text{reject},\text{blank},R) \) except when \( q \) is the accept state. (Any time the TM moves right on a blank in a nonaccepting state, it will never accept.)

We conclude that the modified Turing machine \( M \) is equivalent to an NFA, and hence, it can recognize regular languages. So the language \( L = \{ a^n b^n, n \geq 0 \} \) cannot be accepted.

3.14 Proof:
We need to prove two directions:
(1) Given a deterministic queue automaton DQA, we can simulate Turing machine \( M \) by using the operations defined in DQA.
We write a ‘$’ sign on the queue to indicate the left-hand end of the TM tape. Consider the head of \( M \) always points at the right-hand of the queue. Suppose now \( x \) is on the right-hand end of the queue, then there are two operations of \( M \) to be simulated:

i. \( x \rightarrow y, L \)

We can do this simply by pulling \( x \) out of the queue, and push \( y \) into the queue. This will cycle-shift the content on the queue one bit to the right.

ii. \( x \rightarrow y, R \)

We can write on top of the left-end element in the queue, in order to make it unique. Then, pull out \( x \), and push \( y \) into the queue. We continue to pull out the element on the right-hand end of the queue, but this time we push the same element back to the queue. We repeat this procedure until the dotted element reaches the right-hand end of the queue, then we remove the dot.

In this way, we could simulate the Turing machine by using the operations defined by DQA.
(2) Given a Turing machine \( M \), we can simulate deterministic queue automaton DQA
by using the operations defined in M.
We simulate the queue on the input tape of M. We write ‘#’ on the left-hand end of
the tape, to indicate the left-hand end. There two operations in DQA: push and pull.
   i. push
      In order to push a new element on the left-hand end of the tape, we need to move
      all characters, except ‘#’, on the tape one bit to the right, then we make the head
      of the tape point to the blank right beside ‘#’, and put the element in this position.
   ii. pull
      We could just cross out the elements that needed to be pulled out.
In this way, we could simulate DQA by using Turing machine.
Because of (1) and (2), we say that a language can be recognized by a deterministic
queue automaton iff the language is Turing-recognizable.